Exclusión Mutua para Coordinación de Sistemas Distribuidos

Karina M. Cenci * Jorge R. Ardenghi †

Laboratorio de Investigación en Sistemas Distribuidos
Departamento de Ciencias de la Computación
Universidad Nacional del Sur

Resumen

El problema de exclusión mutua se basa en el acceso a un único e indivisible recurso. Con el auge de los sistemas distribuidos esta problemática se convirtió en una necesidad en diferentes tipos de aplicaciones. Los algoritmos para soportar exclusión mutua en ambientes distribuidos se clasifican en: pasajes de mensajes y datos compartidos (memoria compartida distribuida). El trabajo se basa en algoritmos distribuidos de memoria compartida asincrónica. A partir del algoritmo de Tournament se desarrolla un protocolo de simple escritura y múltiple lectura. Obteniendo ventajas en el diseño e implementación del mismo, ya que es mucho más costoso el acceso a variables compartidas de múltiple escritura que a variables compartidas de simple escritura.

1 Introducción

En los sistemas distribuidos existen un conjunto de procesos que interactúan con el fin de resolver un problemas. En muchos casos, existen vínculos entre estos procesos para trabajar en forma cooperativa; una manera de compartir el trabajo es utilizando recursos compartidos, como pueden ser dispositivos, código, datos, etc. Algunos de los recursos que son compartidos, en un determinado instante de tiempo pueden ser solamente accedidos por un número limitado de procesos, el caso más conocido que sólo un proceso pueda tener acceso a ese recurso, denominado exclusión mutua.

El estudio de la exclusión mutua es el problema de manejar el acceso a un único e indivisible recurso (como por ejemplo una impresora) que solamente puede soportar a un usuario a la vez, entre n usuarios $U_1..U_n$, o los conflictos resultantes de varios procesos concurrentes compartiendo recursos. Alternativamente, se puede pensar éste como el problema de asegurar que ciertas secciones de código de programa sean ejecutadas en forma estrictamente exclusiva (sección crítica).

Un usuario con acceso al recurso es modelado estando en la región crítica, la cual es simplemente un subconjunto de sus estados posibles.

Cuando un usuario no está involucrado de ninguna manera con el recurso, se dice que

*e-mail: kmc@cs.uns.edu.ar
†e-mail: jra@cs.uns.edu.ar

está en la región *resto*. Para obtener la admisión a la región crítica, un usuario ejecuta un protocolo de entrada (*trying*), después que utiliza el recurso, se ejecuta un protocolo de salida (*exit*). Este procedimiento puede repetirse, de modo que cada usuario sigue un ciclo, desplazándose desde la región resto, a la región de entrada, luego a la región crítica y por último a la región de salida, y luego vuelve a comenzar el ciclo en la región resto.

En este trabajo se presenta un protocolo de exclusión mutua para un ambiente de memoria compartida distribuida asincrónica basado en el algoritmo de Tournament, con la utilización de variables de control para acceder a la región crítica de simple escritura y múltiple lectura.

2 Algoritmo de Tournament

El algoritmo de Tournament permite resolver el problema de exclusión mutua para n procesos; está basado en el algoritmo de exclusión mutua de Peterson para dos procesos. Para simplificar, se asume que n es el número de procesos, y es una potencia de 2. La numeración de los procesos comienza en 0 y termina en n-1, en vez de 1 hasta n.

Cada proceso está ocupado en una serie de competiciones de O(log n) para obtener el recurso. Puede pensarse que la competición está dispuesta en un árbol de competencia binario de n hojas, las n hojas corresponden de izquierda a derecha a los procesos 0 .. (n-1).

Se trabaja en un completo árbol binario. Las hojas corresponden a los n procesos. Se utilizan en el algoritmo las siguientes funciones.

- comp (i,k) → en el nivel k del proceso i, es la cadena que consiste de los bits de mayor orden del log n - k de la representación binaria de i. (es el padre).
- role (i,k) → el rol del proceso i en el nivel k de competición del proceso i, es el bit del (log n - k +1) de la representación binaria de i.
- oponentes (i,k) → los oponentes del proceso i en el nivel k de competición del proceso i, es el conjunto de índices de procesos con el mismo orden mayor de bits en log n - k y el opuesto bit en (log n - k + 1).

El algoritmo sería:

Variables compartidas

Para cada cadena binaria x de a lo sumo longitud log n - 1:

Turn(x) ∈ 0,1, inicialmente arbitraria, escrito y leído por exactamente aquellos procesos i para los cuales x es un prefijo de la representación binaria de i.

Para cada i, 0 i n-1:

Flag(i) ∈ 0,..., log n, inicialmente 0, escrita por i y leída por todo j ≠ i

Proceso *i*:

*** Región Resto ***

try i

para k=1 hasta log n hacer
flag(i) := k {representa a los diferentes niveles}
turn(comp(i,k)) := role(i,k)
waitfor [∀ j ∈ oponentes(i,k) : flag(j) < k] o
[turn(comp(i,k)) ≠ role (i,k)]

crit i

*** Región Crítica ***

exit i

flag(i):= 0

rem i

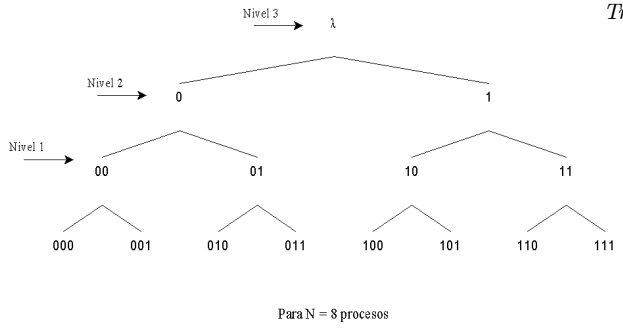
El algoritmo de Tournament satisface exclusión mutua con la propiedad de libre de interbloqueo y de progreso.

Una propiedad poco atractiva del algoritmo de Tournament es que utiliza como variable/dato compartido (*turn*) de múltiple es-

critura/múltiple lectura. Este tipo de variables son difíciles y costosas de implementar en muchas clases de sistemas de multiprocesador como ocurre en sistemas de pasaje de mensajes. Es mejor diseñar algoritmos que utilicen solamente simple escritura y múltiple lectura en sus variables/datos, o aún mejor, simplemente escritura y simple lectura en sus variables datos (pero reduce la concurrencia).

¿Qué sucedería sino fuera potencia de 2?

El árbol binario no sería completo esto implica que para los procesos denominados i con valor cercano a n en algunos niveles no tendría oponentes. En la figura se observa un árbol binario completo para 8 procesos / nodos. En el caso que hubiera sólo 6 procesos / nodos en el segundo nivel para los procesos 5 y 6 no tienen oponentes y pueden pasar al próximo nivel de competencia.



Para N = 8 procesos

Si el proceso desciende de la rama derecha siempre tendrá oponentes sino dependerá del nivel en que se encuentre.

Si se extiende el algoritmo para que N no sea potencia de 2, se debe calcular los niveles que presenta el algoritmo, ya que log N no será un número entero sino que en la mayoría de los casos será un número real.

Se consideran las siguientes definiciones para las funciones que se utilizan en el algoritmo.

- comp (i,k) → el nivel k del proceso i, es la cadena que consiste de los (etapas - k) bits de mayor orden de la representación binaria de i.
- Etapas → está representando la cantidad de bits necesaria para almacenar hasta el valor (n-1)
- Role(i,k) → el rol del proceso i en el nivel k de competición del mismo, es el bit (etapas - k +1) de la representación binaria de i. (representa si desciende de la rama derecha o izquierda).
- Oponentes(i,k) → los oponentes del proceso i en el nivel k de competición del proceso i, es el conjunto de índices de procesos con el mismo orden de bits en (etapas - k) y el opuesto en (nivel - k + 1)

La modificación afectaría de la siguiente manera al algoritmo:

Try i

```
Si truncar(log(n)) = log (n) entonces
  etapas = truncar(log(n))
Sino
  etapas = truncar(log(n)) + 1
Fin si
{etapas contiene la cantidad de niveles}
para k=1 hasta etapas hacer
  flag(i) := k {representa a los diferentes niveles}
  turn(comp(i,k)) := role(i,k)
  si role(i,k) ≠ 0 ó (i     n-k) entonces
    waitfor [∀ j∈ oponentes(i,k) : flag(j) < k] o
    [turn(comp(i,k)) ≠ role (i,k)]
```

---

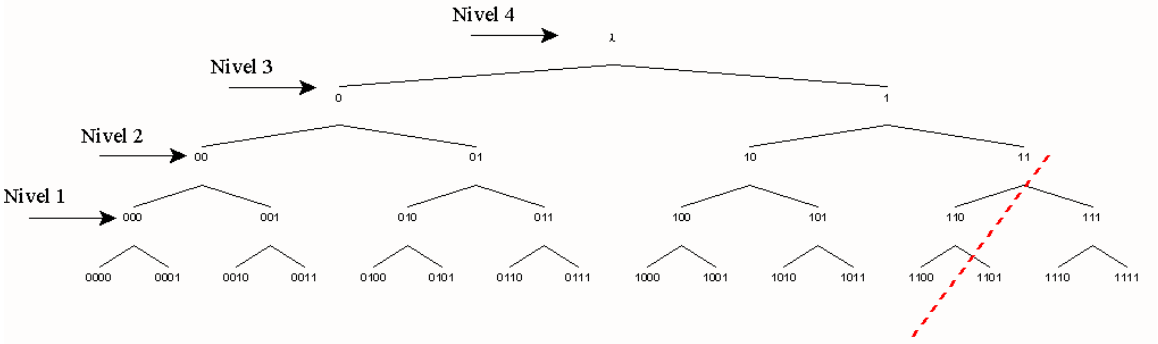
3 Algoritmo Propuesto (simple escritura / múltiple lectura)

La utilización de variables de control de múltiple escritura para el obtener el acceso

a la región crítica son muy costosas e ineficientes de implementar. La utilización de variables de control de simple escritura para el acceso a la sección crítica facilita el diseño e implementación de los protocolos.

En la redefinición del algoritmo y en su transformación a simple escritura y múltiple lectura, no es necesario utilizar la variable de control *flag* para acceder a la sección crítica.

En la próxima figura se muestra un ejemplo de árbol en el cual se considera que se tiene un árbol con 4 niveles con un total de 16 procesos, considerando que se tienen solamente 13 procesos en el sistema. Se puede observar que el proceso 13 no tendr oponentes en los niveles 1 y 2 de competencia. En el algoritmo se considera que tenga oponentes reales.



Árbol completo para 16 procesos, en este caso N=13

El algoritmo con las modificaciones sería el siguiente:

Variables compartidas

Para cada cadena binaria x de a lo sumo longitud etapas -1 y
Para cada i, 0 i n-1
Turn(x,i) ∈ Z ∪ Null, escrita por i y leída por todo j ≠ i

Proceso i

*** Región Resto ***

try i

si truncado(log(n)) = log(n) entonces
etapas := truncado(log(n)) {etapas contiene la cantidad de niveles}
sino
etapas := truncado(log(n)) + 1
fin si
para k=1 hasta etapas hacer
turn(comp(i,k),i) := role(i,k)
si role(i,k) ≠ 0 ó (i n-k) entonces
(∀ j ∈ oponentes(i,k)
si turn(comp(i,k),j) ≠ Null entonces
turn(comp(i,k),i) := turn(comp(i,k),i) + turn(comp(i,k),j)))
waitfor (∀ j ∈ opact(i,k):
(turn(comp(i,k),j)=Null) ó
(turn(comp(i,k),i)<turn(comp(i,k),j)) ó
turn(comp(i,k),i)= turn (comp(i,k),j) y
(i > j)))

*** Región Crítica ***

Exit i

turn(comp(1, 1..etapas), i):= Null

Rem i

Es conveniente modelar el algoritmo en un modelo formal, se lo transforma en un autómata de entrada salida, requiriendo solamente el testeo los oponentes activos en cada uno de los niveles. Se agrega la definición de OpAct(i,k).

- OpAct(i,k) → los oponentes del proceso i en el nivel k de competición del proceso i, es el conjunto de índices de procesos con el mismo orden de bits en (etapas - k) y el opuesto en (nivel - k + 1) y que Turn(x,j) ≠ Null

---

Variables compartidas

Para cada cadena binaria x de longitud a lo sumo etapas -1 y

Para cada i, 0 i n-1:

Turn(x,i) $\in$ 0,1 $\cup$ Null, inicialmente Null

Acciones de i

Entrada	Internas
Try_i	$set-turn_i$
$Exit_i$	$check-oponentes_i$, $\forall$ j $\in$ oponentes(i,nivel)
Salida	$check-turn(j)_i$, $\forall$ j $\in$ opact(i,nivel)
$Crit_i$	$reset_i$
Rem_i	

Estados de i

estado $\in$ {rem, set-turn, check-oponentes, check-turn, leave-try, crit, leave-exit} inicialmente rem

Nivel $\in$ {1 ... etapas}, inicialmente 1

S, un conjunto de índices, inicialmente 0

O, un conjunto de índices, inicialmente oponentes(i, nivel)

A, un conjunto de índices, inicialmente 0

Transiciones de i

Try_i
- efecto:
  - estado := set-turn

$set-turn_i$
- precondición:
  - estado = set-turn1
- efecto:
  - Turn(comp(i,Nivel),i) := role(i,Nivel)
  - If (role(i,Nivel) $\neq$ 0) or (i n-nivel) then
    - //tiene oponentes en el nivel
    - S:= $\emptyset$
    - estado := check-oponentes
  - Else // no tiene oponentes en el nivel
    - If (Nivel < etapas) then
      - Nivel := Nivel + 1
      - O := oponentes(i,Nivel)
    - Else
      - estado := leave-try

$check-oponentes(j)_i$
- precondición:
  - estado = check-oponentes
  - j $\in$ O
  - j $\notin$ S
- efecto:
  - S = S $\cup$ j
  - If Turn(comp(i,Nivel),j) $\neq$ Null then
    - Turn(comp(i,Nivel),i):= Turn(comp(i,Nivel),i)+Turn(comp(i,Nivel),j)
  - else
    - A = A $\cup$ j
    - // Son los procesos que no compiten en el nivel
  - If |S| =|O| then
    - If |A| = |O| then
      - S:=$\emptyset$
      - If Nivel < etapas then
        - Nivel := Nivel + 1
        - O := oponentes(i,Nivel)
        - estado := set-turn
      - Else
        - estado := leave-try
    - Else S:= $\emptyset$ $\cup$ A estado := check-turn

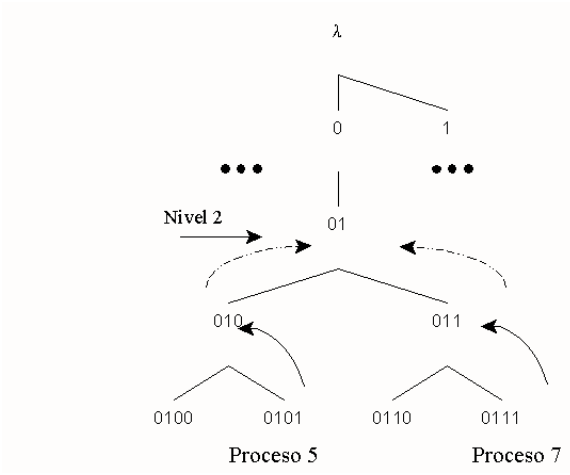
$check-turn(j)_i$
- precondición:
  - estado = check-turn
  - j $\in$ O
  - j $\notin$ S
- efecto:
  - if ((Turn(comp(i,Nivel), j) = Null) or ((Turn(comp(i,Nivel), i)<(Turn(comp(i,Nivel),j)) or ((Turn(comp(i,Nivel),i)=(Turn(comp(i,Nivel),j) and (i >j)) then
    - S := S $\cup$ j
    - If |S| = |O| then
      - S:= $\emptyset$
      - If Nivel < etapas then
        - Nivel := Nivel + 1
        - O := oponentes(i,Nivel)
        - estado := set-turn
      - Else
        - estado := leave-try
  - else
    - S:= $\emptyset$ $\cup$ A

$Crit_i$
- precondición:
  - estado = leave-try
- efecto:
  - estado := crit

$Exit_i$
- efecto:
  - estado := reset

$Reset_i$
- precondición:
  - estado = reset
- efecto:
  - For k :=1 to etapas do
    - Turn(comp(i,k), i) := Null
  - Nivel := 1
  - estado := leave-exit

Rem_i
- precondición:
  - estado := leave-exit
- efecto:
  - estado := rem



En la figura se observa que los procesos 5 y 7 son los ganadores en el primer nivel de competencia y compiten en el segundo nivel en donde son oponentes respectivamente. El estado de las variables, después de pasar por la transición *set-turn*, es el siguiente:

Proceso 5

$Turn(\text{comp}(5,2)) = \text{role}(5,2) = 0$

Proceso 7

$Turn(\text{comp}(7,2)) = \text{role}(7,2) = 1$

Cuando un proceso es el ganador de un nivel (en este caso el primero) vuelve al estado de *set-turn*. Se setea el valor de la variable de control *Turn* correspondiente al nivel que se encuentra. En este estado se controla si el proceso tienen oponentes en el nivel.

- Si no hay oponentes en el nivel se lo considera el ganador y pasa al siguiente.
- Si tiene oponentes, entonces pasa al estado *check-oponentes*. En este estado se chequean los oponentes que se encuentran activos, y luego se pasa al estado *check-turn* hasta que sea el ganador del nivel o pueda acceder a la sección crítica. En el estado que controla *turn* sólo espera por los procesos que son oponentes activos del nivel.

Si es el ganador del nivel, pasa nuevamente al estado *set-turn* y repite el proceso. En el caso que sea el ganador del último nivel pasa al estado *leave-try* que le permite acceder a la sección crítica cambiando el estado a *crit*.

Para el ejemplo de la figura: Los procesos oponentes para el proceso 5 en el nivel 2 son los procesos 6,7. Para el proceso 7 en el nivel 2 son los procesos 4 y 5. Para el proceso 5, en el estado *check-oponentes* se obtiene que sólo el proceso 7 es el oponente activo para su nivel de competencia y ocurre lo mismo para el proceso 7. Al entrar ambos procesos en el estado *check-turn*, habrá un ganador (el proceso 5 ó el proceso 7).

En el caso que sea el proceso 5 el ganador se darían las siguientes condiciones:

$Turn(\text{comp}(5,2)) = 0$

$Turn(\text{comp}(7,2)) = 1$

Al entrar en el estado $check - turn_5(7)$ se verificaría que $turn(\text{comp}(5,2)) < turn(\text{comp}(7,2))$ y por lo tanto se convierte en el ganador del nivel.

En el caso que sea el proceso 7 el ganador se darían las siguientes condiciones:

$Turn(\text{comp}(5,2)) = 1$

$Turn(\text{comp}(7,2)) = 1$

Al entrar en el estado $check - turn_7(5)$ se verificaría que $turn(\text{comp}(7,2)) = turn(\text{comp}(5,2))$ y $(7 > 5)$ por lo tanto se convierte en el ganador del nivel.

4 Consideraciones

Para una dada colección de usuarios (Ui) y para un sistema de memoria compartida A resolver el problema de exclusión mutua significa satisfacer las siguientes condiciones:

- Buena Formación: en cualquier ejecución, para cualquier i, la sub-secuencia

que describe la interacción entre Ui y A está bien formada para i.

- Exclusión Mutual: no se alcanza un estado del sistema (una combinación de un estado del autómata de A y estados para todos los Ui) en el cual más de un usuario se encuentra en la región crítica C.

- Progreso: en cualquier punto de una ejecución imparcial:

  1. (progreso para la región de entrada): si al menos un usuario está en T y ningún usuario en C, en un punto posterior en el tiempo algún usuario entra a C.
  2. (progreso para la región de salida): si al menos un usuario está en E, en un punto posterior en el tiempo algún usuario entre a R.

Un algoritmo bueno de exclusión mutua debería satisfacer:

- Libre de Interbloqueo: cuando la sección crítica (región crítica) está disponible, los procesos no deben esperar indefinidamente y alguno pueda entrar.

- Libre de inanición: cada requerimiento a la sección crítica debería ser eventualmente garantizado.

- Imparcialidad: los requerimientos serán otorgados basados en ciertas reglas de imparcialidad. Típicamente, está basado en los requerimientos de tiempo determinados por relojes lógicos.

El algoritmo presentado está bien formado, ya que el modelo de autómata de I/O presenta esta propiedad. Para la exclusión mutua, la idea clave está en que el nivel k de competición solamente permita un proceso ganador desde cualquier raíz del subárbol en el nivel k.

En cualquier estado del sistema del algoritmo planteado, un proceso i es un ganador en el nivel k si se mantiene:

$nivel_i > k$ ó

$Turn(comp(i, nivel), i) < Turn(comp(i, nivel), j)$ ó

$Turn(comp(i, nivel), i) = Turn(comp(i, nivel), j)$ y $(i > j)$ ó

$nivel_i = k$ y $estado_i \in$ {leave-try, crit, exit}

La última condición sucederá cuando $nivel_i =$ etapas. Un proceso i es un competidor en el nivel k, si es un oponente en el nivel k y el $nivel_i =$ k con $estado_i \in \{$*check-oponentes, check-turn*$\}$.

La clave del algoritmo para mantener exclusión mutua es que exista un solo ganador de cada subárbol por nivel. Supongamos que se verifica hasta el nivel (k-1) y en el nivel k hay dos ganadores entonces:

$Nivel_i = Nivel_j =$ k, en este caso hay un oponente que intenta obtener el acceso. En la verificación de la variable Turn puede suceder lo siguiente:

$Turn(comp(i, k), i) < Turn(comp(i, k), j)$, el proceso i pasa al siguiente nivel y el proceso j debe esperar.

$Turn(comp(i, k), i) = Turn(comp(i, k), j)$, como i≠j, ó i>j ó j>i. Por ende uno de los dos procesos avanza al siguiente nivel y el otro debe esperar.

En los diferentes casos se contradice la suposición inicial de que hay más de un ganador por nivel en cada subárbol. Por lo tanto, el algoritmo satisface exclusión mutua.

Si en una ejecución a, hay por lo menos un proceso en la región de entrada y no hay

ningún proceso en la región crítica, supongamos que no entra ningún proceso. Si el proceso, denominado i, está en la región de entrada entonces está esperando en el estado de check-turn, pero al no haber ningún proceso en la región crítica, entonces $nivel_j < nivel$ (se obtiene del chequeo de *turn*(comp(i,k),j) =Null), para i≠j entonces podrá avanzar de nivel y en el último nivel tendrá la posibilidad de acceder a la región crítica. Esto contradice la suposición inicial.

Si un algoritmo está bien formado y es libre de bloque (para todos los procesos) entonces garantiza la propiedad de progreso. Si mantiene la propiedad de vivacidad entonces no puede ocurrir inanición. ¿Es posible que un proceso espere indefinidamente para acceder a la sección crítica? Cuando un proceso i ingresa en la región de entrada, pasa por el estado *set-turn*, en el cual inicializa la variable *turn* para ese nivel y en el estado *check-oponentes* controla cuales procesos son oponentes activos, y actualiza el valor de *turn* de acuerdo a los oponentes activos y al orden de llegada, por lo tanto un proceso j que accede después a la región de entrada, deberá esperar que el proceso i tenga su acceso a la región crítica o al próximo nivel antes de poder avanzar.

5 Conclusión

La coordinación de procesos distribuidos requieren del soporte de exclusión mutua. El ambiente de los sistemas distribuidos puede estar basado en Memoria Compartida Distribuida o mediante la utilización de mensajes. En este trabajo, se consideró el entorno de Memoria Compartida Distribuida, utilizando variables de control compartida para acceder a la sección crítica.

El modelo presentado, basado en el algoritmo de Tournament, se extiende a un número arbitrario N de procesos, sin la restricción de que sea una potencia de 2. Se utiliza una variable de control Turn de simple escritura / múltiple lectura. El algoritmo está bien formado, por el modelo de Autómata de I/O, garantizando la exclusión mutua y el progreso. La ventaja del algoritmo presentado está en la implementación ya que es más fácil diseñar y codificar variables compartidas de simple escritura y múltiple lectura en ambientes distribuidos; aún a costa de la reducción en la concurrencia en el acceso a los datos compartidos. En la región de entrada se testea en cada uno de los niveles sólo los procesos oponentes que se encuentran activos en el momento de la competencia.

Referencias

[1] Jie Wu, *Distributed System Design*, 1999.

[2] Gary L. Peterson, Myths about the mutual exclusion problem. *Information Processing Letters*, Junio 1981.

[3] Nancy A Lynch. *Distributed Algorithms*, 1997.

[4] Sape Mullender. *Distributed Systems*, 2da. Ed. 1993.

[5] Michael Raynal. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, 1986.

[6] M. Ben Ari. *Principles of Concurrent Programming*. Prentice Hall, Englewood Cliffs, 1982.